CSC 273 – Data Structures

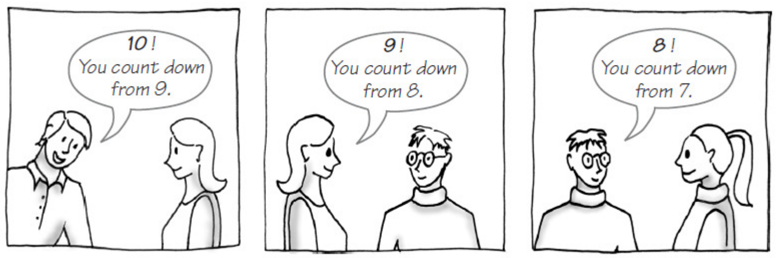
Lecture 4- Recursion

What Is Recursion?

- Consider hiring a contractor to build
  - He hires a subcontractor for a portion of the job
  - That subcontractor hires a sub-subcontractor to do a smaller portion of job
- The last sub-sub- … subcontractor finishes
  - Each one finishes and reports “done” up the line

Example: The Countdown

Counting down from 10

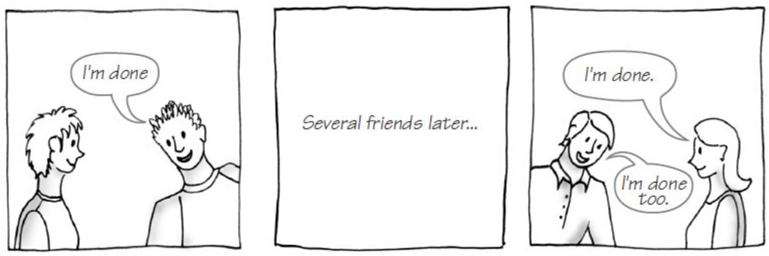


Example: The Countdown

Counting down from 10

Example: The Countdown

Counting down from 10



Recursive **countdown()**

```
public static void countDown(int integer) {
      System.out.println(integer);
      if (integer > 1)
           countDown(integer - 1);
  }
```

Definition

- Recursion is a problem-solving process
  - Breaks a problem into identical but smaller problems.
- A method that calls itself is a ***recursive method*.**
  - The invocation is a ***recursive call*** or ***recursive invocation***.

Design Guidelines

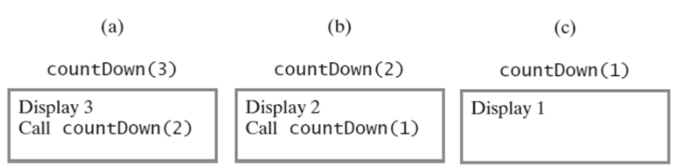
- Method must be given an input value
- Method definition must contain logic that involves this input, leads to different cases
- One or more cases should provide solution that does not require recursion
  - Else infinite recursion
- One or more cases must include a recursive invocation

Programming Tip

- Iterative method contains a loop
- Recursive method calls itself
- Some recursive methods contain a loop and call themselves
  - If the recursive method with loop uses **`while`**, make sure you did not mean to use an **`if`** statement

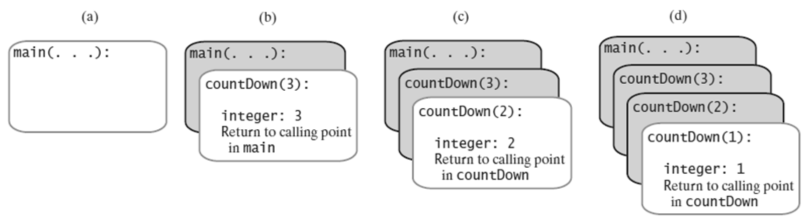
Tracing a Recursive Method

The effect of the method call **`countDown(3)`**



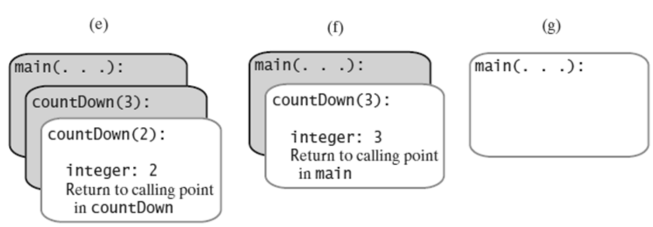
Tracing a Recursive Method

The stack of activation records during the execution of the call **`countDown(3)`**



Tracing a Recursive Method

The stack of activation records during the execution of the call **`countDown(3)`**



Stack of Activation Records

- Each call to a method generates an activation record
- Recursive method uses more memory than an iterative method
  - Each recursive call generates an activation record
- If recursive call generates too many activation records, could cause stack overflow

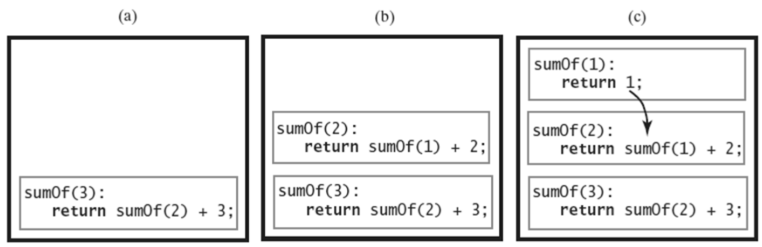
Recursive Methods That Return a Value

Recursive method to calculate $\sum_{i=1}^{n} i$

```
public static void countDown(int integer) {
      System.out.println(integer);
      if (integer > 1)
          countDown(integer - 1);
}
```

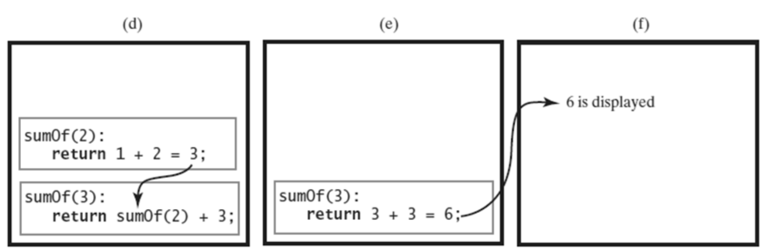
Tracing a Recursive Method

Tracing the execution of **sumOf(3)**



Tracing a Recursive Method

Tracing the execution of **sumOf(3)**



Recursively Processing an Array

Given definition of a recursive method to display array

```
// Displays the integers in an array.
// Array - an array of integers
// first -  the index of the first element
//          displayed
// last - the index of the last element display
//    0 <= first <= last < array.length
public static void displayArray
          (int [] array, int first, int last)
```

Recursively Processing an Array

```
public static void displayArray
         (int [] array, int first, int last) {
      System.out.print(array[first] + " ");
      if (first < last)
          displayArray(array, first + 1, last);
}
```

Starting with **`array[first]`**

Recursively Processing an Array

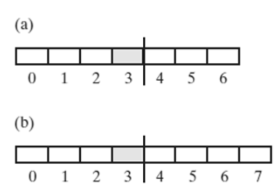
```
public static void displayArray
         (int [] array, int first, int last) {
                  if (first < last)
      displayArray(array, first, last - 1);

      System.out.print(array[last] + " ");
}
```

Starting with **`array[last]`**

Recursively Processing an Array

```
int mid = (first + last) / 2;
```



Two arrays with their middle elements within their left halves

Recursively Processing an Array

```
public static void displayArray
        (int array[], int first, int last) {
   if (first == last)
      System.out.print(array[first] + " ");
   else {
      int mid = (first + last) / 2;
      displayArray(array, first, mid);
      displayArray(array, mid + 1, last);
   }
}
```

Processing array from middle

Displaying a bag

```
/*
 *  display() - displays the contents of an array
 *              bag using the recursive method
 *              displayArray
 */
 public void display() {
   displayArray(0, numberOfEntries - 1);
 }

 private void displayArray(int first,int last) {
   System.out.println(bag[first]);
   if (first < last)
     displayArray(first + 1, last);
 }
```

Recursively Processing a Linked Chain

```
public void display()
{
   displayChain(firstNode);
} // end display
private void displayChain(Node nodeOne)
{
   if (nodeOne != null)
   {
      System.out.println(nodeOne.getData()); // Display first node
      displayChain(nodeOne.getNextNode());   // Display rest of chain
   } // end if
} // end displayChain
```

Display data in first node and recursively display data in rest of chain.

Recursively Processing a Linked Chain

```
public void displayBackward()
{
   displayChainBackward(firstNode);
} // end displayBackward
private void displayChainBackward(Node nodeOne)
{
   if (nodeOne != null)
   {
      displayChainBackward(nodeOne.getNextNode());
      System.out.println(nodeOne.getData());
   } // end if
} // end displayChainBackward
```

Displaying a chain backwards. Traversing chain of linked nodes in reverse order easier when done recursively.

Time Efficiency of Recursive Methods

```
public static void countDown(int n)
{
   System.out.println(n);
   if (n > 1)
      countDown(n - 1);
} // end countDown
```

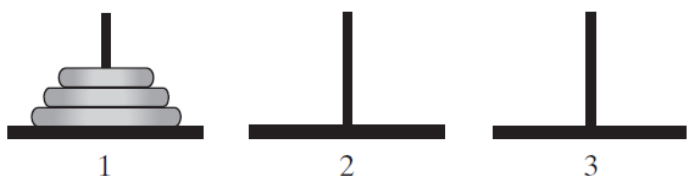
Using proof by induction, we conclude method is O(n)

Time Efficiency of Computing x^n

$x^n = (x^{n/2})^2$ when n is even and positive
$x^n = x\,(x^{(n-1)/2})^2$ when n is odd and positive
$x^0 = 1$

Efficiency of algorithm is O(log n)

Simple Solution to a Difficult Problem



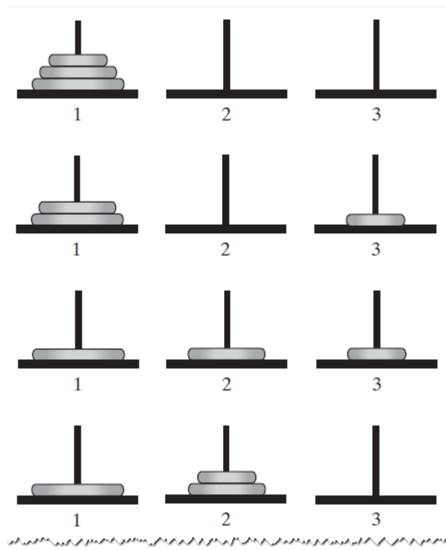
The initial configuration of the Towers of Hanoi for three disks.

Simple Solution to a Difficult Problem

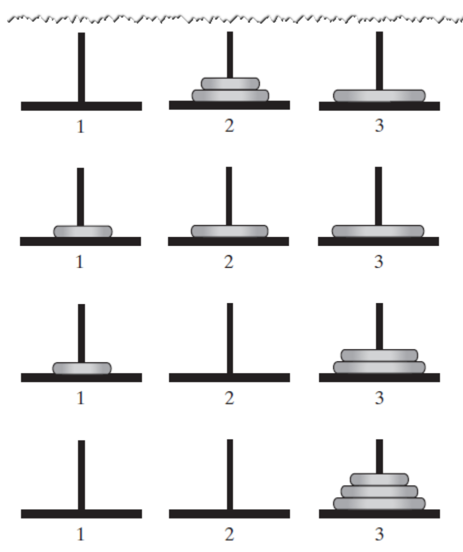
<u>Rules:</u>

1. Move one disk at a time. Each disk moved must be topmost disk.
2. No disk may rest on top of a disk smaller than itself.
3. You can store disks on the second pole temporarily, as long as you observe the previous two rules.

The sequence of moves for solving the Towers of Hanoi problem with three disks



The sequence of moves for solving the Towers of Hanoi problem with three disks



Solutions

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
   Move disk from startPole to endPole
else
{
   solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
   Move disk from startPole to endPole
   solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

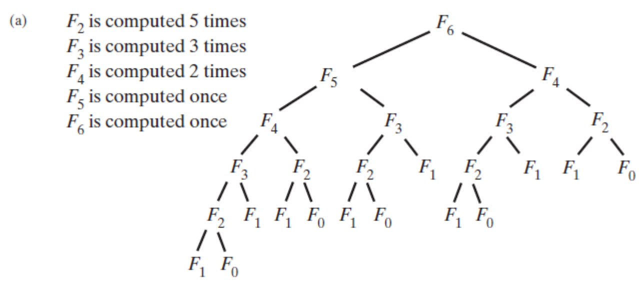
Recursive algorithm to solve any number of disks.
Note: for n disks, solution will be $2^n - 1$ *moves*

Poor Solution to a Simple Problem

```
Algorithm Fibonacci(n)
if (n <= 1)
   return 1
else
   return Fibonacci(n - 1) + Fibonacci(n - 2)
```

Algorithm to generate Fibonacci numbers.
Why is this inefficient?

Poor Solution to a Simple Problem



The computation of the Fibonacci number F_6 using recursion

(b)

$$
\begin{aligned}
F_0 &= 1\\
F_1 &= 1\\
F_2 &= F_1 + F_0 = 2\\
F_3 &= F_2 + F_1 = 3\\
F_4 &= F_3 + F_2 = 5\\
F_5 &= F_4 + F_3 = 8\\
F_6 &= F_5 + F_4 = 13
\end{aligned}
$$

The computation of the Fibonacci number F_6 using iteration.

Poor Solution to a Simple Problem

Tail Recursion

```
public static void countDown(int integer)
{
   if (integer >= 1)
   {
      System.out.println(integer);
      countDown(integer - 1);
   } // end if
} // end countDown
```

Tail recursion

When the last action performed by a recursive method is a recursive call

Tail Recursion

- In a tail-recursive method, the last action is a recursive call
- This call performs a repetition that can be done by using iteration.
- Converting a tail-recursive method to an iterative one is usually a straightforward process.

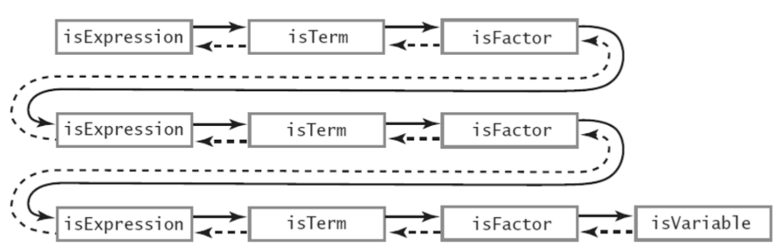
Indirect Recursion

- Example
  - Method A calls Method B
  - Method B calls Method C
  - Method C calls Method A
- Difficult to understand and trace
  - But does happen occasionally

Indirect Recursion

- Consider evaluation of validity of an algebraic expression
  - Algebraic expression is either a term or two terms separated by a + or – operator
  - Term is either a factor or two factors separated by a * or / operator
  - Factor is either a variable or an algebraic expression enclosed in parentheses
  - Variable is a single letter

Indirect Recursion – An Example



Replacing Recursion with Iteration

```
public void displayArray(int first, int last)
{
   if (first == last)
      System.out.println(array[first] + " ");
   else
   {
      int mid = first + (last - first) / 2; // Improved calculation of midpoint
      displayArray(first, mid);
      displayArray(mid + 1, last);
   } // end if
} // end displayArray
```

Using a Stack Instead of Recursion

```
private void displayArray(int first, int last)
{
   boolean done = false;
   StackInterface<Record> programStack = new LinkedStack<Record>();
   programStack.push(new Record(first, last));
   while (!done && !programStack.isEmpty())
   {
      Record topRecord = programStack.pop();
      first = topRecord.first;
      last = topRecord.last;
```

Using a Stack Instead of Recursion

```
      if (first == last)
         System.out.println(array[first] + " ");
      else
      {
         int mid = first + (last - first) / 2;
         // Note the order of the records pushed onto the stack
         programStack.push(new Record(mid + 1, last));
         programStack.push(new Record(first, mid));
      } // end if
   } // end while
} // end displayArray
```